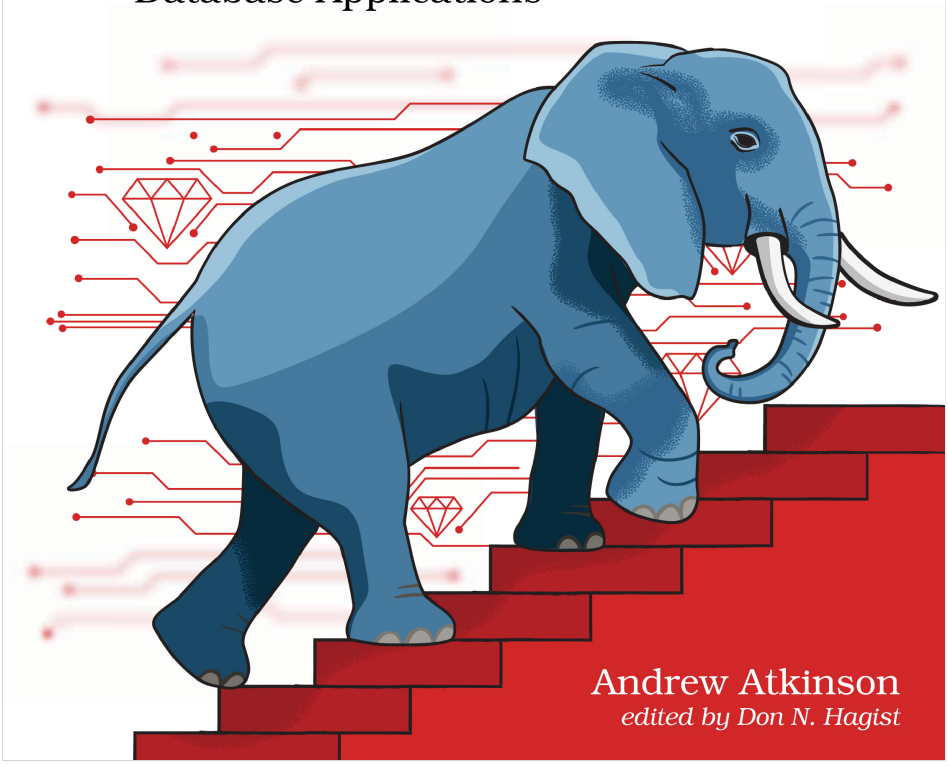
The Pragmatic Programmers

High Performance PostgreSQL for Rails

Reliable, Scalable, Maintainable Database Applications



Andrew Atkinson
edited by Don N. Hagist

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 7

Improving Query Performance

In this chapter, we'll focus on SQL queries and factors that influence their performance.

You'll learn about query execution plans and how to read them, identifying the most costly parts. With an understanding of the costs, you'll learn tactics to lessen them, speeding up queries and reducing use of system resources.

Query optimization is a complex subject with entire books dedicated to it. In this chapter, you'll get set up with the basics.

Review the following terminology you'll see in upcoming sections:

Query Performance Terminology

- Selectivity—How narrow or wide a selection is
- Cardinality—How many unique values there are
- Sequential scan—Reading all rows for a table, also called *table scan*
- Index scan—Fetching values from an index
- Index-only scan—Fetching values *only* from the index, without needing to access table data

Although you'll primarily deal with SQL and PostgreSQL in this chapter, let's start out by discussing slow query visibility in Active Record.

How can you find slow queries?

Active Support Instrumentation for Queries

Without adding extra Ruby gems or PostgreSQL extensions, you can capture slow queries to the Rails log. *Active Support Notifications*[1] are a mechanism

1. https://guides.rubyonrails.org/active_support_instrumentation.html

that emit events with event data. The relevant events here are sql.active_record events.

How does that work? Take a look at the following slow query *Subscriber* class, which was added to Rideshare.

The Subscriber listens for sql.active_record events, calculating a query duration from the start and finish values. When the query takes more than one second, the query text is logged.

ruby/slow_query_subscriber.rb

```
# Inspiration: https://twitter.com/kukicola/status/1578842934849724416
class SlowQuerySubscriber < ActiveSupport::Subscriber
  SECONDS_THRESHOLD = 1.0

  ActiveSupport::Notifications.subscribe('sql.active_record')
    do |name, start, finish, _, data|
    duration = finish - start

    if duration > SECONDS_THRESHOLD
      Rails.logger.debug "[#{name}] #{duration} #{data[:sql]}"
    end
  end
end
```

Open bin/rails console to test this out. Run SELECT PG_SLEEP(1); within ActiveRecord::Base .connection.execute() to create a query that will take one second. You'll see the Subscriber is triggered, and the query is logged with sql.active_record prepended:

ruby/active_record_slow_query_subscriber.rb

```
ActiveRecord::Base.connection.execute("SELECT PG_SLEEP(1)")
[sql.active_record] 1.008904 SELECT PG_SLEEP(1) /*application='Rideshare'*/
   (1009.2ms)  SELECT PG_SLEEP(1) /*application='Rideshare'*/
```

While this technique can be used for the Rails log, how might you capture slow queries in PostgreSQL?

Capture Query Statistics in Your Database

The queries in your database consume resources. You'll want to optimize them to be less costly, focusing your optimization efforts on the biggest beneficiaries.

To find costly queries and make data-driven decisions, you'll need a global view of all queries and their statistics. To do that, use the pg_stat_statements[2] module you configured earlier (see Modifying Your PostgreSQL Config File, on page ?), which we'll abbreviate PGSS.

2. https://www.postgresql.org/docs/current/pgstatstatements.html

PGSS performs a normalization process for each query, removing specific parameters and replacing their values with placeholder characters (question marks).

The normalized query gets a *query identifier* (queryid), which represents a query group. Similar normalized queries placed into the same group are grouped together. Statistics are collected at the group level. PGSS presents the statistics in a catalog view that you can enable access to for your database.

More than 40 fields of information are collected as statistics[3] for PGSS. Some of the information includes the number of calls for queries within that group and their execution time min, max, mean, and standard deviation. These statistics are cumulative, growing until less-used query groups are evicted or statistics are reset. Rows and blocks that are accessed are included in the stats, which can be used to help identify excessive IO.

Since you added PGSS to shared_preload_libraries in postgresql.conf and restarted PostgreSQL, we'll assume it's ready to be used.

To make the system view available, connect to the rideshare_development database as the postgres superuser:

```
psql -U postgres -d rideshare_development
```

From there, run the following statement to create the extension within the rideshare schema:

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements
WITH SCHEMA rideshare;
```

Editing Config File

ALTER SYSTEM can modify shared_preload_libraries as an alternative to editing postgresql.conf. This method generates a value in postgresql.auto.conf that overrides the value in postgresql.conf.

To avoid confusion about where the active value originates, skip ALTER SYSTEM and edit postgresql.conf directly.

You've now enabled PGSS and are ready to use it.

Using Query Statistics

Since Rideshare isn't a running system, you'll need to simulate application activity so that query statistics can be calculated from it.

3. https://www.postgresql.org/docs/current/pgstatstatements.html

Start the Rideshare server by running bin/rails server in your terminal.

In another terminal window, run the Rake task:

sh/simulate_app_activity.sh
```
bin/rails simulate:app_activity
```

As queries are received in PostgreSQL, PGSS places them into groups, gives the group an identifier, and captures group-level statistics. PGSS tracks 5000 normalized queries (or query groups) by default, which can be increased by setting pg_stat_statements.max.

The least-executed queries are discarded when the max is reached. To reset the statistics, run SELECT rideshare.PG_STAT_STATEMENTS_RESET(); from psql.

Great. If you reset the statistics, run the simulation again. Once you've done that, you should now have some stats to work with. Let's use the stats to find some of the ten slowest queries by mean execution time:

sql/ten_worst_queries.sql
```
SELECT
  mean_exec_time,
  calls,
  query,
  queryid
FROM pg_stat_statements
ORDER BY mean_exec_time DESC
LIMIT 10;
```

Great, you're able to view the PGSS information in psql and see some of the worst-performing queries.

Use pspg pager

pspg pager can be added to make the wide query results like the ones from PGSS more legible. With pspg configured as the pager, you're able to navigate horizontally within psql.

Refer to the Development Guides[4] repository or db/README.md in Rideshare for instructions and usage.

Rideshare queries should be displayed in descending order. An example result is shown as follows:

4. https://github.com/andyatkinson/development_guides

sql/pg_stat_statements_result.sql

```
-[ RECORD 8 ]--+-----------------------------------------------------
mean_exec_time | 2878.5335836666663
calls          | 3
query          | SELECT COUNT(*) FROM "users" \
                   WHERE "users"."type" = $1 /*application:Rideshare*/
queryid        | 5435614976858805274
```

In this example, we can see the mean_exec_time, the number of calls, the query text, and the queryid.

While viewing statistics from psql works, you'd like to make this information more accessible to your team.

How can you do that?